STRUKTURALNI PATERNI

1. Adapter patern

Adapter patern služi da se postojeći objekat prilagodi za korištenje na neki novi način u odnosu na postojeći rad, bez mijenjanja same definicije objekta. Na taj način obezbjeđuje se da će se objekti i dalje moći upotrebljavati na način kako su se dosad upotrebljavali, a u isto vrijeme će se omogućiti njihovo prilagođavanje novim uslovima.

Ukoliko bismo uveli korištenje loyalty kartice na kojoj se skupljaju bodovi, onda bi mogli iskoristiti Adapter patern, da pretvorimo te bodove u popust.

2. Facade patern

Fasadni patern služi kako bi se klijentima pojednostavilo korištenje kompleksnih sistema. Klijenti vide samo fasadu, odnosno krajnji izgled objekta, dok je njegova unutrašnja struktura skrivena. Na ovaj način smanjuje se mogućnost pojavljivanja grešaka jer klijenti ne moraju dobro poznavati sistem kako bi ga mogli koristiti.

EParking nam je fasada, unutar koje se obavljaju procesi u koje korisnik nema uvid (npr. plaćanje preko metode *platiParking* koja u sebi poziva nake druge implementirane metode interfejsa *IPayPal* i *ICreditCard*; parkiranje preko metode *izracunajRutu* koja poziva neke implementirane metode intefejsa *IGoogleMaps*).

3. Decorator patern

Decorator patern služi za omogućavanja različitih nadogradnji objektima koji svi u osnovi predstavljaju jednu vrstu objekta (odnosno, koji imaju istu osnovu). Umjesto da se definiše veliki broj izvedenih klasa, dovoljno je omogućiti različito dekoriranje objekata (tj. dodavanje različitih detalja), te se na taj način pojednostavljuje i rukovanje objektima klijentima, i samo implementiranje modela objekata.

Trenutno nemamo potrebu za primjenom ovog paterna. Ukoliko bi se odlučili da primijenimo ovaj patern, mogli bismo dodati mogućnost slanja obavještenja ne samo preko E-Maila, nego i preko SMS-a, Facebook-a, Viber-a, WhatsApp-a.

4. Bridge patern

Bridge patern služi kako bi se apstrakcija nekog objekta odvojila od njegove implementacije. Ovaj patern veoma je važan jer omogućava ispunjavanje Open-Closed SOLID principa, odnosno uz poštivanje ovog paterna omogućava se nadogradnja modela klasa u budućnosti te osigurava da se neće morati vršiti određene promjene u postojećim klasama.

U našem sistemu, primijenili smo Bridge patern kod obračuna cijena u zavisnosti od tipa korisnika. Gost plaća cijenu po satu, a član u zavisnosti od tipa članarine. Kreirali smo

interface *IPlacanjeBridge* koji u sebi sadrži metodu *obracunajCijenu*, i tu metodu svaka od klasa implementira po svojim potrebama.

5. Composite patern

Composite patern služi za kreiranje hijerarhije objekata. Koristi se kada svi objekti imaju različite implementacije nekih metoda, no potrebno im je svima pristupati na isti način, te se na taj način pojednostavljuje njihova implementacija.

Ako bi u našem sistemu, klasu *Administrator* učinili apstraktnom i iz nje naslijedili klase *AdministratorParkinga* i *AdministratorKorisnika*, tada bi dodali interface *IAdministrator* sa metodom *azuriraj*, koju bi svaka od izvedenih klasa implementirala na svoj način.

6. Proxy patern

Proxy patern služi za dodatno osiguravanje objekata od pogrešne ili zlonamjerne upotrebe. Primjenom ovog paterna omogućava se kontrola pristupa objektima, te se onemogućava manipulacija objektima ukoliko neki uslov nije ispunjen, odnosno ukoliko korisnik nema prava pristupa traženom objektu.

U našem sistemu primijenili smo Proxy patern kod pregleda analitike prihoda od strane vlasnika parkinga. Dodali smo klasu *IzvjestajProxy* koja u sebi sadrži instancu interfejsa *IIzvjestajProxy* koja ima metodu *pregledajAnalitikuPrihoda*. Na osnovu atributa *kodParkinga* ograničeno je da svaki vlasnik može pregledati analitiku prihoda svog parkinga.

7. Flyweight patern

Flyweight patern koristi se kako bi se onemogućilo bespotrebno stvaranje velikog broja instanci objekata koji svi u suštini predstavljaju jedan objekat. Samo ukoliko postoji potreba za kreiranjem specifičnog objekta sa jedinstvenim karakteristikama (tzv. specifično stanje), vrši se njegova instantacija, dok se u svim ostalim slučajevima koristi postojeća opća instanca objekta (tzv. bezlično stanje). Korištenje ovog patterna veoma je korisno u slučajevima kada je potrebno vršiti uštedu memorije.

Nismo implementirali ovaj patern, ali mogli bismo ga primijeniti kada bismo dodali klasu *ClanFirme*, u slučaju da se svi radnici neke firme parkiraju na istom parkingu i samim tim imaju određeni popust. Kada bi se član neke firme prijavljivao na sistem, ne bismo svaki put kreirali novu instancu klase *ClanFirme*, već bi se instancirao samo jedan objekat ovog tipa i tako bismo uštedili memoriju.

KREACIJSKI PATERNI

1. Singleton patern

Singleton patern služi kako bi se neka klasa mogla instancirati samo jednom. Na ovaj način može se omogućiti i tzv. lazy initialization, odnosno instantacija klase tek onda kada se to prvi put traži. Osim toga, osigurava se i globalni pristup jedinstvenoj instanci - svaki put kada joj se pokuša pristupiti, dobiti će se ista instanca klase. Ovo olakšava i kontrolu pristupa u slučaju kada je neophodno da postoji samo jedan objekat određenog tipa.

Ovaj patern je primijenjen na klasi *EParking*, jer je potrebna samo jedna instanca ove klase. Dodali smo metodu *dajInstancu* u klasu *EParking*.

2. Prototype patern

Prototype patern omogućava smanjenje kompleksnosti kreiranja novog objekta tako što se uvodi operacija kloniranja. Na taj način prave se prototipi objekata koje je moguće replicirati više puta a zatim naknadno promijeniti jednu ili više karakteristika, bez potrebe za kreiranjem novog objekta nanovo od početka. Ovime se osigurava pojednostavljenje procesa kreiranja novih instanci, posebno kada objekti sadrže veliki broj atributa koji su za većinu instanci isti.

Mi smo u našem sistemu implementirali ovaj patern, tako što smo dodali interface *IPrototip*, koji ima metodu *kloniraj*. Time smo pojednostavili proces kreiranja novih instanci objekta, a klasa koja ga implementira je *Transakcija.*

3. Factory Method patern

Factory method patern služi za omogućavanje instanciranje različitih vrsta podklasa koristeći factory metodu koja odlučuje koja će se podklasa instancirati i koja programska logika izvršiti. Na ovaj način osigurava se ispunjavanje O SOLID principa, jer se kod za kreiranje objekata različitih naslijeđenih klasa ne smješta samo u jednu zajedničku metodu, već svaka podklasa ima svoju logiku za instanciranje željenih klasa, a samo instanciranje kontroliše factory metoda koju različite klase implementiraju na različit način.

Mi nismo u našem sistemu implementirali ovaj patern. Ukoliko bismo ga željeli implementirati, mogli bismo dodati klasu *AnalitikaPrihoda* iz koje su naslijeđene klase *GodisnjiPrihodi* i *MjesecniPrihodi*, a klasa *IzvjestajPrihoda* bi u sebi imala atribut *analitikaPrihoda*. Dodali bismo i dvije izvedene klase *StariIzvjestajPrihoda* i *NoviIzvjestajPrihoda* (izvedene iz klase *IzvjestajPrihoda*), koje implementiraju interface *IIzvjestaj* koji sadrži metodu *kreirajIzvjestaj* (factory metoda).

4. Abstract Factory patern

Abstract factory patern služi kako bi se izbjeglo korištenje velikog broja if-else uslova pri kreiranju različitih hijerarhija objekata. Ukoliko postoji više tipova istih objekata te različite klase koriste različite podtipove, te klase postaju fabrike za kreiranje objekata zadanog podtipa bez potrebe za specificiranjem pojedinačnih objekata. Na ovaj način se, korištenjem nasljeđivanja, ukida potreba za postojanjem if-else uslova jer određeni tip fabrike sadrži određene tipove objekata i zna se tačno koju podklasu će instancirati.

U našem sistemu nismo implementirali ovaj patern. Navedeni patern bismo mogli implementirati ukoliko bismo dodali interface *IFactory* sa metodom *dajNacinPlacanja*, čime bismo smanjili broj if-else uslova za određivanje načina plaćanja. Mi smo trenutno to riješili korištenjem enumeracije *NacinPlacanja*.

5. Builder patern

Builder patern služi za apstrakciju procesa konstrukcije objekta, kako bi se kao rezultat mogle dobiti različite specifikacije objekta koristeći isti proces konstrukcije. Ovaj patern koristi se kako bi se izbjeglo kreiranje kompleksne hijerarhije klasa te kako bi se izbjegao kompleksni programski kod konstruktora jedne klase koja može imati različite konfiguracije atributa. Različiti dijelovi konstrukcije objekta izdvajaju se u posebne metode koje se zatim pozivaju različitim redoslijedom ili se poziv nekih dijelova izostavlja, kako bi se dobili željeni različiti podtipovi objekta bez potrebe za kreiranjem velikog broja podklasa.

Trenutno nismo implemenitrali ovaj patern. Ukoliko bismo uzimali u obzir i parkinge za veća vozila (npr. kamione , autobuse) , tada bismo mogli omogućiti da se na početku bira tip vozila, i na mapi bi se prikazale informacije samo o tim parkinzima. Dodali bismo i interface *IBuilder* koji sadrži metode koje će izvršavati prikaz samo određenih parkinga, kao i klasu *TipVozilaBuilder* koja implementira metode interfejsa.